DATAMEMO: TURNING ACQUIRED WEB DATA INTO REUSABLE AGENT MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Web analytical tasks require agents to collect, join, and compute over records dispersed across web sources. When successive requests concern the same source, reacquiring those records wastes time and tokens, while retaining them only in unstructured interaction history makes coverage and interpretation difficult to track. Despite these challenges, dedicated benchmarks and systematic studies of how agents should manage and reuse web data remain limited. To evaluate this setting, we introduce DATAMEMOBENCH, a benchmark comprising 3,799 questions across eight domains to evaluate agents on successive analytical requests over shared web sources. We also present DATAMEMO, a relational agentic data memory framework that enables agents to incrementally store extracted records in a relational database, reuse them across questions, and acquire additional data as needed. Our evaluation shows that a continued-session baseline retains prior dialogue and files without explicitly representing the coverage and interpretation of retained data, while episodic and procedural memory still incur substantial overhead from repeated data acquisition and processing. With GPT-5.5 and Claude Opus 4.8, DATAMEMO reduces token cost by 36–46% and wall-clock time by 6–13%, while achieving a 5.8–11.8% relative improvement in accuracy over the strongest baseline, with efficiency gains growing as more questions reuse the accumulated data. Together, DATAMEMOBENCH and DATAMEMO provide an evaluation testbed and an effective framework for accurate and efficient web analysis.

1 INTRODUCTION

Web agents are increasingly used to perform analytical tasks beyond basic question answering and information retrieval (Deng et al., 2023; Zhou et al., 2024). These tasks require collecting records and deriving answers through filtering, comparison, and aggregation, often across related questions over the same source. Information gathering and analysis already constitute a substantial share of existing evaluations: our analysis of 3,992 tasks across seven web-agent benchmarks finds that 61.8% require gathering, comparing, or aggregating information as their final deliverable (Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024; He et al., 2024; Pan et al., 2024; Yoran et al., 2024; Xue et al., 2025). However, analytical questions in these benchmarks are blended with generic retrieval and question-answering tasks and often involve only small collections or simple computations, providing limited insight into how agents handle successive, data-intensive analytical requests (Zhou et al., 2024; Miyai et al., 2025).

This limited coverage also leaves weaknesses in current agent frameworks insufficiently examined, particularly their **ability to manage acquired data efficiently for future analytical requests over the same sources**. For example, answering "Which institutions would be most affected by ICLR's new per-author submission limits?" may require assembling thousands of paper, author, and affiliation records. Without direct access to the full dataset, the agent must first discover and collect relevant information, then use specialized API queries to check subsets of records within the service's access limits. Over time, users may pose many successive questions about authors, institutions, and submission patterns, each requiring different subsets of records and attributes. This creates a retention problem: keeping only task-specific findings can discard information useful for future analysis, while carrying all accumulated observations and intermediate results in context becomes increasingly costly. Existing agentic memory techniques provide limited guidance on what acquired data to retain, at what granularity, and how to make it available for future computations without continually

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

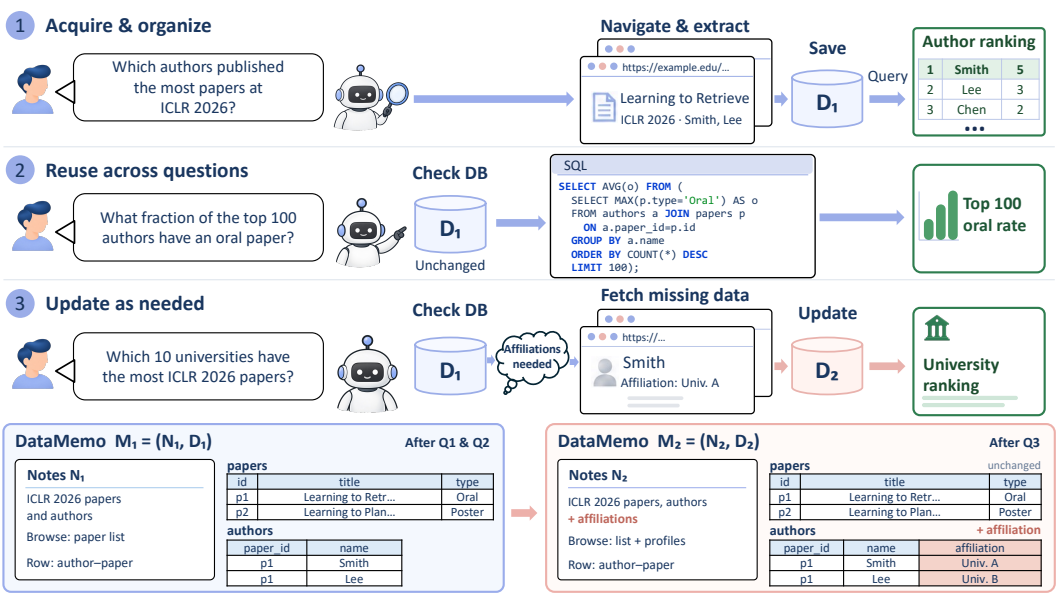


Figure 1: **Overview of DATAMEMO.** Across successive questions over a shared web source, the agent **(1)** acquires and organizes records into relational memory, **(2)** reuses stored data for new analyses, and **(3)** collects missing information and updates memory as needed.

expanding the context footprint (Packer et al., 2023; Xu et al., 2026b; Yu et al., 2026). Consequently, agents may repeat costly acquisition and processing, especially under specific site constraints (e.g., API rate limits) or incur overhead gathering usable data from growing interaction histories. Both approaches risk incomplete or inconsistent answers: 53% to 84% of errors stem from data grounding failures, such as incomplete crawls or misread field definitions (§5.2). These limitations motivate better evaluation and technical support for reliable, efficient data-intensive web analysis.

In this paper, we address these evaluation and technical gaps through DATAMEMOBENCH, a dedicated benchmark, and DATAMEMO, a framework for managing acquired data across analytical requests. Specifically, DATAMEMOBENCH organizes diverse, self-contained analytical questions around shared real-world web sources, enabling controlled evaluation of data reuse across successive requests. We select 28 websites across eight domains from Online-Mind2Web (Xue et al., 2025) that support data-intensive analysis, with sources ranging from approximately 1K to 30M records. For each website, we construct 109–204 analytical questions paired with ground-truth reference answers, yielding 3,799 questions evaluated as successive requests over shared sources. These questions cover retrieval, filtering, aggregation, and more complex compositions, alongside unanswerable probes that test whether agents recognize insufficient source information. To establish verifiable and reproducible reference answers, our construction pipeline separates question design from answer computation: questions are developed around persona-guided information needs, while reference answers are computed programmatically from validated data acquired from the source websites and then subjected to separate automated checks and human review.

We also introduce DATAMEMO as a framework that treats data acquired during exploration as an agent-managed resource rather than transient interaction context (Figure 1). At test time, agents incrementally construct, query, and maintain a relational store of extracted records, accompanied by notes. The store allows agents to solve new tasks by reusing previously acquired data; if necessary, agents can collect new information and update it to support future reuse. This design trades persistent storage for reduced data acquisition and context processing, while preserving record-level values for future analysis. We evaluate DATAMEMO against no memory, episodic memory, procedural memory, and continued sessions baseline methods across two base models. DATAMEMO improves accuracy by 5.8–11.8% relative to the strongest baseline while significantly reducing token cost by 36–46%, with efficiency gains increasing as successive requests arrive. On the held-out analytical tasks from WebArena (Zhou et al., 2024) and WebChoreArena (Miyai et al., 2025), DATAMEMO reduces wall-clock time by 50–58% and cost by 30–48% relative to baseline methods.

Our contributions are threefold: We introduce DATAMEMOBENCH, a benchmark for evaluating agents' accuracy and efficiency across successive, data-intensive analytical requests over shared web sources. We introduce DATAMEMO, a relational agentic data memory framework that enables agents to retain, reuse, and maintain acquired records for future analysis. Finally, we demonstrate accuracy and efficiency gains across two base models on DATAMEMOBENCH and external benchmarks.

2 RELATED WORK

2.1 BENCHMARKS FOR WEB AND DATA ANALYSIS AGENTS

WebArena and WebChoreArena evaluate interaction and information processing in controlled web environments (Zhou et al., 2024; Miyai et al., 2025), while Online-Mind2Web evaluates task completion on live websites (Xue et al., 2025). WebLists, WideSearch, and DeepWideSearch emphasize structured extraction, broad information collection, and multi-hop reasoning (Bohra et al., 2025; Wong et al., 2026; Lan et al., 2025). Data analysis benchmarks support complex computation but typically begin with supplied datasets or databases, leaving web acquisition outside their main task scope (Lei et al., 2025; Jing et al., 2025; Egg et al., 2025). More recent works started to bridge these two directions: WebAggregatorQA emphasizes cross-source evidence gathering and combination (Wang et al., 2026), and WebDS evaluates broader data science workflows that include the downstream actions (Hsu et al., 2026). Their evaluations focus on individual task completion. Existing benchmarks cover complementary aspects of web analysis (Table A.1).

These benchmarks touch on web analytical tasks within broader task suites, but provide limited dedicated evaluation of data-intensive, successive analytical requests over shared web sources. DATAMEMOBENCH addresses this gap with 3,799 questions across eight domains, organized around shared sources to evaluate how agents manage and reuse acquired data for different analytical demands.

2.2 MEMORY FOR LLM AGENTS

Agent memory systems retain interaction histories, consolidate facts, and organize linked knowledge (Packer et al., 2023; Chhikara et al., 2025; Xu et al., 2026b; Yang et al., 2026). Other approaches reuse trajectory exemplars (Zheng et al., 2024), induce workflows and executable skills (Wang et al., 2024; Zheng et al., 2025; Prabhu et al., 2026), or maintain reasoning strategies and evolving insights (Ouyang et al., 2026; Suzgun et al., 2026; Zhang et al., 2026; Xu et al., 2026a). These approaches improve access to prior knowledge and experience, but their retained content may not preserve the records needed for new analytical computations. Procedures can guide acquisition while still requiring repeated collection, adding tool calls and latency. Memory maintenance introduces further overhead: under matched token budgets, a vanilla web agent can match or outperform online skill and memory modules (Hajimiri et al., 2026). Facts and summaries may omit records or fields needed by subsequent filters and aggregations, risking incomplete or incorrect answers. Related evidence shows that repeated context rewriting can erase useful details and degrade task performance (Zhang et al., 2026). Retaining full interaction histories avoids some information loss but increases token costs and latency (Chhikara et al., 2025), while leaving agents vulnerable to overlooking relevant evidence within long contexts (Liu et al., 2024).

Structured memory provides a complementary foundation. ChatDB uses SQL databases as symbolic memory (Hu et al., 2023), and Table-Augmented Generation combines language-model reasoning with computation over existing data (Biswal et al., 2024). On the acquisition side, BardeenAgent reuses extraction programs across similarly structured pages (Bohra et al., 2025). However, these memory approaches primarily preserve interaction history, task-specific findings, or reusable procedures rather than the underlying records needed for future analysis, making successive data-intensive requests costly to support. Our DATAMEMO focuses on managing and reusing the acquired records themselves across analytical requests. Its agent-managed relational store supports new computations over retained data, with further acquisition and revision as needed. This reduces repeated collection and context processing while preserving record-level values that summaries may omit. By querying retained data without reprocessing full interaction histories, DATAMEMO supports data reuse beyond procedural guidance and avoids the context overhead of carrying forward entire trajectories.

3 METHOD

In this section we formalize the problem setting (§3.1) and present DATAMEMO (§3.2). §B.1 defines the memory conditions under this interface and a continued-session reference condition. Figure 1 gives an overview of the full pipeline.

3.1 PROBLEM FORMULATION

Agent configuration. We study an LLM-based agent $\mathcal{A} = (\Phi, \mathcal{M})$: a frozen LLM backbone Φ and an on-disk memory store $\mathcal{M}$. The agent acts through an action space $\mathcal{U}$: it can execute code in a sandbox shell, and it can search, visit, and fetch pages on the live site $\mathcal{S}$. The memory is reached through a memory interface staged into the working directory and called like any other program, so memory operations are also configured as actions in $\mathcal{U}$. The memory interface exposes the same three operations under every condition: READ($\mathcal{M}$) returns the short summary of the memory that is placed in the prompt; QUERY($\mathcal{M}, x$) returns the part of the memory matching x without loading the rest; and WRITE($\mathcal{M}, x$) adds, revises, or deletes. Conditions differ in which programs realize these operations, and hence in what $\mathcal{M}$ may hold, detailed in §B.1.

Task setting. As agents take on long-running, persistent roles, they naturally encounter a continuous stream of tasks (Ouyang et al., 2026), and a website serving analytical needs likewise receives successive requests from one or many users and agents. We model this naturally occurring workload as a stream of questions about one website $\mathcal{S}$, revealed online in batches $B_1, \ldots, B_T$ (five questions each in our experiments). Each batch is answered in a fresh agent session that holds all its questions in a single prompt, but none of the context of earlier batches. Under each memory condition, $\mathcal{M}$ is the only channel between batches. A batch is the unit of work:

$$\big(B_k,\ \mathcal{M}_{k-1}\big) \longmapsto \big(A_k,\ \mathcal{M}_k\big), \qquad A_k = \big\{(q,\ \hat{a}_q,\ s_q,\ \mu_q)\big\}_{q \in B_k}, \tag{1}$$

with $\hat{a}_q$ the answer or $\varnothing$, $s_q \in \{answered, unanswerable, error\}$ and μ_q the method used; all answers are returned together, and only $\hat{a}_q$ is graded.

3.2 DATAMEMO

Our memory has two parts: a relational *data memory* $\mathcal{D}$ holding the records, and a thin *note layer* $\mathcal{N}$ holding the rules for using them correctly.

Memory schema. $\mathcal{D}$ is a set of typed base relations in SQLite, one per entity kind the site exposes. Each multi-valued field is normalized into a child relation, so that aggregation is defined over individual values rather than over a packed string. Each relation includes metadata describing its coverage and fetch timestamps, along with agent-authored probes for checking the source for changes. $\mathcal{N}$ is a set of atomic rules covering how to access the site and how to match, deduplicate, and count. Its purpose is to let future agents re-derive answers rather than store them.

Read. READ($\mathcal{M}$) returns $\mathcal{N}$ and a compact digest of $\mathcal{D}$, including table schemas, row counts, coverage claims, and fetch timestamps. The records themselves stay on disk and are reached only through queries. From the second batch onward, this digest helps the agent identify reusable data and decide whether further acquisition or freshness checks are needed. Writes to $\mathcal{D}$ happen via code execution rather than text rewrite, so stored records do not drift through the agent's paraphrase.

Memory update: hits, misses, and refreshes. For each question, the agent assesses *coverage* and *freshness*. These checks serve different purposes: coverage determines whether the store contains the required data, while freshness determines whether source changes warrant updating them. The agent first checks *coverage* within the store: whether the stored entities, fields, and keys cover the question's scope, using coverage claims and notes on failed acquisition. Missing required data constitute a **miss**; the agent fetches them from $\mathcal{S}$, incorporates them into $\mathcal{D}$, and updates the coverage claim. Independently of coverage, the agent checks freshness when source changes could affect the answer. It probes the source by comparing record counts for population changes, record IDs for additions and removals, and sampled records for changes in field values. Detected changes mark the affected data as **stale**, triggering a **refresh** before answering. The agent adds or updates records,

Algorithm 1 DATAMEMO over a batch stream. Each batch is processed in a fresh session. The inner loop describes per-question logic.

Require: Φ; site $\mathcal{S}$; batches $B_1, \ldots, B_T$
Ensure: $A_k = \{(q, \hat{a}_q, s_q, \mu_q)\}_{q \in B_k}$ for each k; memory $\mathcal{M}_T$

$(\mathcal{D}, \mathcal{N}) \leftarrow (\varnothing, \varnothing)$
for $k = 1, \ldots, T$ **do**
  start a session with READ$(\mathcal{D}, \mathcal{N})$ and all of B_k ▷ no context from $B_{<k}$
  for all $q \in B_k$ **do**
    if $\mathcal{D}$ does not cover q **then** ▷ **miss**
      $\mathcal{D} \leftarrow$ WRITE$(\mathcal{D},$ ACQUIRE$(\mathcal{S}, q))$
    end if
    if source changes could affect q **and** PROBE$(\mathcal{D}, \mathcal{S}, q)$ detects changes **then**
      $\mathcal{D} \leftarrow$ WRITE$(\mathcal{D},$ REFRESH$(\mathcal{S}, q))$ ▷ **stale**
    end if
    $\hat{a}_q \leftarrow$ QUERY$(\mathcal{D}, q)$
    determine status s_q and record method μ_q
    $\mathcal{N} \leftarrow$ WRITE$(\mathcal{N},$ rules learned on $q)$
  end for
  $A_k \leftarrow \{(q, \hat{a}_q, s_q, \mu_q)\}_{q \in B_k}$
  $\mathcal{M}_k \leftarrow (\mathcal{D}, \mathcal{N})$
  output A_k
end for

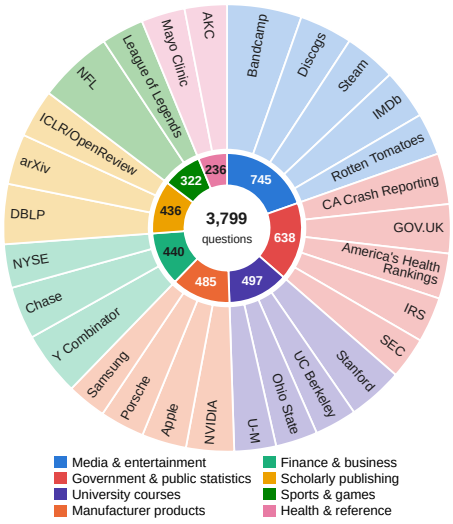


Figure 2: Distribution of 3,799 questions across eight domains.

Table 1: Personas in the ICLR testbed and example questions from the "Peer-review process researcher" persona.

Personas: Peer-review process researcher; University research-office analyst; Science-policy analyst; AI industry research strategist; Conference program chair; PhD student scouting the field.

Question	Reference answer
How does the acceptance rate vary as the spread between the highest and lowest reviewer ratings increases from 0 to 8?	For spreads 0, 2, 4, 6, and 8: 17.9%, 26.2%, 28.4%, 33.5%, and 41.1%.
Among submissions with mean ratings in $[5, 6)$, are those with high reviewer disagreement (SD ≥ 1.5) accepted less often than those with low disagreement (SD < 1.0)?	Yes: 60.1% versus 76.5%.
What numeric overall recommendation did the area chair assign to "The Polar Express: Optimal Matrix Sign Methods and their Application to the Muon Algorithm"?	Unavailable: the published meta-review contains text fields but no numeric recommendation.
What are the lowest mean rating among accepted submissions and the highest mean rating among rejected submissions?	2.50 and 7.00, respectively.

but deletes them only when a stable source listing covering the same population confirms their absence. Records whose absence remains uncertain are retained, with this uncertainty noted in the coverage claim. A question is a **hit** when the stored data already provide sufficient coverage, and the agent judges their freshness adequate after any necessary checks. Once coverage and freshness are adequate, the agent answers through SQL. The agent retains updated data, check results, and timestamps for later batches. It also updates notes with newly learned rules for accessing data or interpreting fields. Algorithm 1 summarizes this process.

4 DATAMEMOBENCH

We introduce DATAMEMOBENCH to evaluate agent memory across successive analytical requests over shared web sources. It comprises 3,799 questions across 28 websites in eight domains, in-

cluding product catalogs, public statistics, and scholarly publications (as shown in Figure 2). Each testbed specifies a website and a set of analytical questions with reference answers. Questions are individually self-contained but draw on overlapping data, allowing agents to reuse previously acquired records for different information needs.

The benchmark contains 1,868 standard questions (49.2%) covering retrieval, filtering, and aggregation; 1,560 hard questions (41.1%) requiring compositions such as nested aggregation, cross-entity reasoning, and per-group ranking; and 371 unanswerable probes (9.8%) requesting information unavailable from the designated source. Table 1 illustrates a sample set of questions using the ICLR website, which covers 19,814 submissions to ICLR 2026 and their reviews and decisions.

Below we discuss the key benchmark construction steps. More details are explained in Appendix A.

Website selection and scope. We manually select testbeds from the websites in Online-Mind2Web (Xue et al., 2025) using four criteria. (1) *Permitted access*: the site allows the automated access needed to acquire its records. (2) *Suitability for data analysis*: the site hosts collections that support comparisons, aggregations, and relationships across records. (3) *Feasible acquisition*: the collection can be enumerated reproducibly and acquired within the evaluation budget, which we verify with live probes of the available access paths and acquisition throughput. (4) *Stability*: the records change slowly enough for reference answers to remain valid, which excludes, for example, shopping sites with frequently changing prices. This yields 28 websites in eight domains.

Question design and generation. Questions should reflect what users want to know about a website, not what is easy to query from its data, so we write them from personas. For each testbed, Claude Opus 4.8 (Anthropic, 2026) reads a short overview of the site and writes 5 to 10 personas representing plausible user information needs (e.g., "peer-review process researcher" or "conference program chair"; see Table 1). A separate authoring agent then writes questions for these personas from the same overview, without seeing the reference database, its schema, or any answer. Unanswerable probes ask for information that does not exist on the website, testing whether an agent says so instead of hallucinating an answer. Questions and reference answers follow seven design principles (realism, objectivity, clarity, verifiability, context independence, time invariance, and diversity), detailed in Appendix A.2. Appendix A.3 gives the generation prompts.

Reference data and answer construction. Analytical questions in DATAMEMOBENCH are answered by computation over records rather than looking up a single fact. Ground truth therefore requires both correct underlying data and faithful computation. To address these, we first build and validate a reference database, then derive and verify answers from it. For each website, we build a SQLite database with reproducible scripts. Authors manually review the schema, record counts, and relations. This database serves as the ground-truth data layer. Since manual inspection of every record is infeasible at scale, we validate the database against independently published facts: a sourcing agent independently collects 3,508 factual claims from third-party publications (median 4 publishers per testbed). A separate agent then answers these claims using only the database. Agreement confirms that our built database reproduces independently reported facts. Disagreements are manually investigated against the sources to distinguish extraction errors from ambiguous published claims. After corrections on a development split (1,763 pairs), we freeze the database and evaluate the database quality with the held-out test split (1,745 pairs) once, achieving 98.1% accuracy across 21 testbeds. None of the mismatches come from a wrong database value. The authors check the other 7 testbeds manually. These facts serve only to validate the databases and are not benchmark questions (Appendix A.5).

Reference answers are then computed by SQL scripts over the frozen databases. Then a second, independently written computation must agree. Agreement confirms the arithmetic but not the reading of the question, so an agent also answers each question without seeing the reference, which exposes alternative readings, and a premise audit flags ambiguous scope, unclear matching or tie rules, and premises the data contradict. For unanswerable probes, we confirm that they cannot be derived from published fields. Flagged questions are corrected, reworded, or removed, then checked again.

Human review. Finally, the authors review all 28 testbeds for scope, database structure, validation evidence, and question quality. An author also audits 280 randomly sampled questions, ten per testbed, checking each question's clarity and source support and verifying its reference answer with

Table 2: Results on DATAMEMOBENCH, averaged over 28 testbeds (**best**, second best). Cost and time are per testbed. Cost uses list prices per million tokens: $5 input, $0.50 cached input, and $30 (GPT-5.5) or $25 (Opus 4.8) output.

	GPT-5.5			Opus 4.8		
Condition	Acc. ↑	Cost ($) ↓	Time (min) ↓	Acc. ↑	Cost ($) ↓	Time (min) ↓
No memory	0.511	41.23	166	0.297	42.11	195
Procedural memory	0.543	25.57	155	0.515	21.65	167
Episodic memory	0.555	26.10	150	0.360	34.61	181
Continued session	0.693	25.85	107	0.583	33.54	134
DATAMEMO (ours)	**0.733**	**14.06**	**93**	**0.652**	**21.55**	**126**

an independent calculation or query. The audit found no errors in the questions or reference answers; its one flagged inconsistency came from a misreading by the auditor.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

We evaluate agents on all testbeds in DATAMEMOBENCH with Claude Opus-4.8 (Anthropic, 2026) and GPT-5.5 (OpenAI, 2026), through smolagents (Hugging Face, 2025) with tools for web access and code execution. Analytical questions arrive in batches of five. We compare five memory conditions: no memory, procedural memory inspired by ReasoningBank (Ouyang et al., 2026), episodic memory inspired by Synapse (Zheng et al., 2024), DATAMEMO, and a continued-session reference that retains conversation history and the working directory. All conditions receive the same questions in the same shuffled order, with a time budget of three hours per condition. Once the budget is reached, no new batch is started, but the last batch already in progress is allowed to finish. We report accuracy as the fraction of all questions in a testbed that are answered correctly within the budget of three hours. We use GPT-5.5 to score each response as correct or incorrect against its verified reference answer, typically a numerical value or short factual statement. In our evaluation, we explicitly include all tokens, inference costs, and elapsed time required for memory construction and updates in the reported efficiency metrics. Beyond the main experiments, we evaluate generalization on WebArena (Zhou et al., 2024) and WebChoreArena (Miyai et al., 2025), and conduct ablation studies to assess the robustness of DATAMEMO.

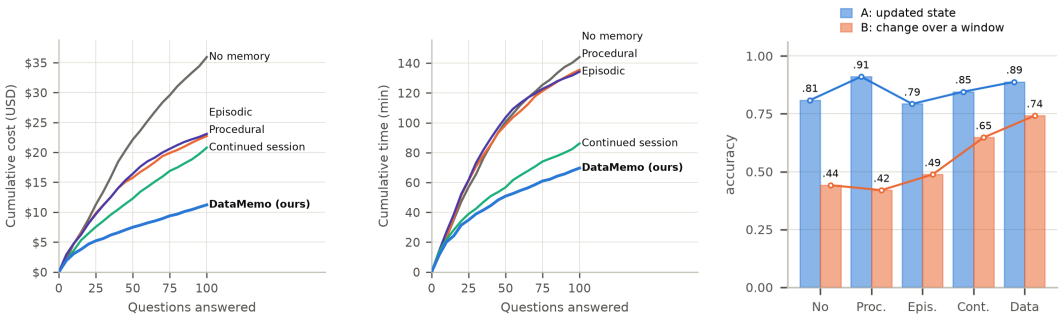
5.2 MAIN RESULTS

DATAMEMO improves accuracy and efficiency across both base models. DATAMEMO achieves the highest mean accuracy, lowest token cost, and shortest wall-clock time on DATAMEMOBENCH (Table 2). Compared with the continued session, the strongest accuracy baseline, it improves accuracy by a relative 5.8–11.8% while reducing cost by 36–46% and time by 6–13%. Other memory baselines are less accurate because they re-derive their data in every batch, so 53–84% of their errors are data grounding mistakes (e.g., an incomplete crawl, a misread definition), against 35% for DATAMEMO. They are also less efficient because the re-fetching dominates their budget, and data that was never fully acquired accounts for up to 40% of their errors, against 14% for DATAMEMO.

Efficiency gains accumulate over successive questions. Figure 3 shows that DATAMEMO accumulates cost and runtime at a slower rate than the baselines on GPT-5.5, with savings growing as more questions are answered. The gap also widens relative to continued sessions, suggesting that naively keeping data and conversation history alone does not provide equally efficient reuse compared to DATAMEMO. As shown in Figure 6, agents with DATAMEMO make substantially fewer web access calls in later batches, once the relational store has been built over the first few batches.

5.3 TIME VARIANCE EXPERIMENTS

In dynamic web environments, previously built memory can become outdated as web content changes. We therefore examine how useful each memory remains for websites whose contents



(a) Cumulative cost. (b) Cumulative time. (c) Time-variance accuracy.

Figure 3: **Efficiency and temporal robustness with GPT-5.5.** (a) Cumulative inference cost and (b) cumulative wall-clock time over the question stream (lower is better). (c) Accuracy on updated-state questions and questions about changes over time (higher is better).

Table 3: Results of time variance questions.

Memory	A: updated	B: change	Overall	Cost ($)	Time (min)
No memory	.81	.44	.613	$32.4	94
Procedural	**.91**	.42	.649	$13.0	78
Episodic	.79	.49	.630	$11.6	44
Continued session	.85	.65	.740	$8.8	**14**
DATAMEMO	.89	**.74**	**.810**	**$4.6**	18

Table 4: External benchmark results.

Eval	Site	Method	Acc.	Time (min) ↓	Cost ($) ↓
WebChore Arena	Shopping	DATAMEMO	**0.673**	**56**	**14**
		ReasoningBank	0.660	114	24
	Shopping Admin	DATAMEMO	0.677	**52**	**13**
		ReasoningBank	**0.710**	125	25
WebArena	Shopping	DATAMEMO	0.523	**64**	**9**
		ReasoningBank	**0.585**	143	15
	Shopping Admin	DATAMEMO	**0.610**	**22**	**7**
		ReasoningBank	0.585	44	10

change over time. For five websites, we construct tasks with the same generator and validation pipeline as the rest of DATAMEMOBENCH and split them by what they require. Part A asks for updated information on the website, and admits a question only when the store built two weeks earlier would answer it incorrectly. Part B asks what changed over a time window. Every condition starts from the memory its own agent built two weeks earlier.

Table 3 and Figure 3c summarize the results with GPT-5.5. On Part A, the form of memory matters little, since rereading the current page is sufficient. On Part B, the conditions separate: DATAMEMO reaches 0.74, whereas procedural (0.42) and episodic (0.49) memory stay near the no memory baseline (0.44), because these memories may omit historical records needed for comparison. The gain of DATAMEMO is largest on sites that publish only their current state, and small on sites that publish dated records (e.g., arXiv), where rereading the source recovers the same facts. These results suggest that the benefit of DATAMEMO comes from retaining timestamped records, which preserve the source's earlier state for comparison and let the agent efficiently refresh only the stale records.

5.4 GENERALIZATION TO EXTERNAL BENCHMARKS

DATAMEMO generalizes to external benchmarks at lower cost. We further compare DATAMEMO with ReasoningBank (Ouyang et al., 2026) on the Shopping and Shopping Admin sites of WebArena (Zhou et al., 2024) and WebChoreArena (Miyai et al., 2025). We select single-site information-retrieval tasks evaluated by string matching that require no screenshots, forming streams of parameterized task instances delivered in batches of five. Both methods use GPT-5.5 with identical task orders, access permissions, and budgets. Table 4 shows that DATAMEMO reduces inference cost by 30–48% and wall-clock time by 50–58% relative to ReasoningBank across all four settings. These results demonstrate that the proposed framework is robust and can be generalized to other forms of benchmarks.

Table 5: **Batch size ablation.** One question per batch (GPT-5.5, 28 testbeds). Δ is the relative change from the five question baseline. Bold marks the best result.

	Accuracy ↑		Cost ($/testbed) ↓		Time (min/testbed) ↓	
Method	Value	Δ	Value	Δ	Value	Δ
No memory	0.293	−42.7%	41.11	−0.3%	186	+12.0%
Procedural memory	0.368	−32.2%	26.11	+2.1%	176	+13.5%
Episodic memory	0.469	−15.5%	31.53	+20.8%	171	+14.0%
Continued session	0.547	−21.1%	46.09	+78.3%	150	+40.2%
DATAMEMO	**0.676**	−7.8%	**24.16**	+71.8%	**121**	+30.1%

Table 6: **Tiered model ablation.** GPT-5.5 for the build batch and GPT-5-mini thereafter (28 testbeds). Gray values are the all-GPT-5.5 results of Table 2. Bold marks the best tiered result.

	Procedural memory	Episodic memory	Continued session	DATAMEMO
Accuracy ↑	0.315 (0.543)	0.230 (0.555)	0.316 (0.693)	**0.476** (0.733)
Accuracy retained ↑	58%	41%	46%	**65%**
Cost ($/testbed) ↓	4.19 (25.57)	4.03 (26.10)	4.15 (25.85)	**3.71** (14.06)
Time (min/testbed) ↓	150 (155)	141 (150)	**68** (107)	117 (93)

Table 7: **Question order ablation.** Mean ± SD over the default order and two random shuffles with GPT-5.5; only the strongest baseline is rerun due to compute limits.

Method	Accuracy ↑	Cost ($/testbed) ↓	Time (min/testbed) ↓
Continued session	0.706 ± 0.021	26.3 ± 0.6	103.7 ± 3.1
DATAMEMO	**0.731 ± 0.005**	**14.8** ± 0.7	**90.7** ± 2.1

5.5 ABLATION STUDIES

DATAMEMO is robust to smaller batches. Reducing the batch size from five questions to one (Table 5) reduces DATAMEMO's accuracy by only 7.8%, versus 15.5% to 42.7% for the baselines, since its stored records carry across batches.

DATAMEMO best preserves accuracy with a smaller answering model. When memory is built with GPT-5.5 and later batches are answered by `gpt-5-mini` (Table 6), DATAMEMO retains the most accuracy (65% vs. at most 58%) at the lowest cost, though its accuracy still drops, showing that reusable data eases but does not remove dependence on the answering model.

The gains of DATAMEMO do not depend on question order. Across the default order and two random shuffles (Table 7; per order results in Appendix C), DATAMEMO beats continued session on every metric in every order and varies less in accuracy, so its gains do not hinge on question order.

6 CONCLUSION

We introduced DATAMEMOBENCH and DATAMEMO to study how web agents can turn acquired data into reusable memory for successive analytical requests. DATAMEMOBENCH makes this capability measurable through diverse, data-intensive questions over shared real-world sources; DATAMEMO supports it through an agent-managed relational store. Across two base models, our approach improves accuracy while reducing cost and latency, with efficiency gains growing as more questions reuse accumulated data. The central finding is that organizing retained records makes future computation more efficient and manageable. Together, our benchmark and framework offer a testbed and a practical approach for improving data reuse in analytical web agents.

AI USE STATEMENT

In accordance with ICLR 2027 policy, we disclose our use of LLMs in preparing this work. We used LLMs to improve the clarity and readability of the manuscript, including refining sentence structure, correcting grammar, and reducing redundant wording. During benchmark construction, LLMs assisted with question generation and automated validation. Recognizing that model-generated content can contain errors, we combined these automated procedures with programmatic cross-checks and human review, as detailed in our benchmark construction protocol. In the experiments, LLMs also served as the evaluated agents and answer judges, with their roles and evaluation procedures described in the paper.

ETHICS STATEMENT

This work does not raise specific ethical concerns. This work uses publicly accessible web sources based on established benchmarks, as described in the paper. It does not involve human subjects, sensitive data, or private information. Our evaluation concerns agents' ability to analyze published information and does not establish the suitability of their outputs for consequential decisions about individuals or organizations.

REPRODUCIBILITY STATEMENT

We will release DATAMEMOBENCH and the code implementation of DATAMEMO, including the evaluation scripts and configuration files, to support reproducibility and future research. We document the memory framework and update procedure, the evaluation protocol, and metrics in the paper. Appendix A provides the benchmark construction procedures, generation prompts, reference-data validation, and answer verification. Appendix B specifies the agent configuration, tools, budgets, question-order seed, and prompts for each memory condition.

REFERENCES

Anthropic. Introducing claude opus 4.8, May 2026. URL https://www.anthropic.com/news/claude-opus-4-8.

Asim Biswal, Liana Patel, Siddarth Jha, Amog Kamsetty, Shu Liu, Joseph E Gonzalez, Carlos Guestrin, and Matei Zaharia. Text2sql is not enough: Unifying ai and databases with tag. *arXiv preprint arXiv:2408.14717*, 2024.

Arth Bohra, Manvel Saroyan, Danil Melkozerov, Vahe Karufanyan, Gabriel Maher, Pascal Weinberger, Artem Harutyunyan, and Giovanni Campagna. Weblists: Extracting structured information from complex interactive websites using executable llm agents. *arXiv preprint arXiv:2504.12682*, 2025.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. Dabstep: Data agent benchmark for multi-step reasoning. *arXiv preprint arXiv:2506.23719*, 2025.

Sina Hajimiri, Masih Aminbeidokhti, Jose Dolz, Ismail Ben Ayed, Issam H Laradji, Spandana Gella, and Nicolas Gontier. Are online skill and memory modules always worth their tokens? a budget-constrained study of web agents. *arXiv preprint arXiv:2606.15017*, 2026.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6864–6890, 2024.

Ethan Hsu, Hong Meng Yam, Ines Bouissou, Aaron John, Raj Thota, Josh Koe, Vivek Putta, G Dharesan, Alexander Spangher, Shikhar Murty, et al. Webds: An end-to-end benchmark for web-based data science. In *International Conference on Learning Representations*, volume 2026, pp. 31777–31795, 2026.

Chenxu Hu, Jie Fu, Chenzhuang Du, Simian Luo, Junbo Zhao, and Hang Zhao. Chatdb: Augmenting llms with databases as their symbolic memory. *arXiv preprint arXiv:2306.03901*, 2023.

Hugging Face. smolagents: a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`, 2025.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. Dsbench: How far are data science agents from becoming data science experts? In *International Conference on Learning Representations*, volume 2025, pp. 32597–32649, 2025.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Tian Lan, Bin Zhu, Qianghuai Jia, Junyang Ren, Haijun Li, Longyue Wang, Zhao Xu, Weihua Luo, and Kaifu Zhang. Deepwidesearch: Benchmarking depth and width in agentic information seeking. *arXiv preprint arXiv:2510.20168*, 2025.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, volume 2025, pp. 28691–28735, 2025.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Atsuyuki Miyai, Zaiying Zhao, Kazuki Egashira, Atsuki Sato, Tatsumi Sunada, Shota Onohara, Hiromasa Yamanishi, Mashiro Toyooka, Kunato Nishina, Ryoma Maeda, et al. Webchorearena: Evaluating web browsing agents on realistic tedious web tasks. *arXiv preprint arXiv:2506.01952*, 2025.

OpenAI. Introducing GPT-5.5: A new class of intelligence for real work, 2026. URL `https://openai.com/index/introducing-gpt-5-5/`.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, et al. Webcanvas: Benchmarking web agents in online environments. *arXiv preprint arXiv:2406.12373*, 2024.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Krithika Ramakrishnan, Jing Gu, Yanqi Luo, Caiming Xiong, Junnan Li, Zeyuan Chen, Ran Xu, et al. Walt: Web agents that learn tools. In *International Conference on Learning Representations*, volume 2026, pp. 55589–55607, 2026.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Rui Wang, Ce Zhang, Jun-Yu Ma, Jianshu Zhang, Hongru Wang, Yi Chen, Boyang Xue, Tianqing Fang, Zhisong Zhang, Hongming Zhang, et al. Webaggregator: Enhancing compositional reasoning capabilities of deep research agent foundation models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24486–24517, 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Ryan Wong, Jiawei Wang, Li Chen, Yan Gao, Xuan Zhou, Zuo Wang, Kai Xiang, Ge Zhang, Wenhao Huang, Yang Wang, et al. Widesearch: Benchmarking agentic broad info-seeking. In *International Conference on Learning Representations*, volume 2026, pp. 10012–10086, 2026.

Weijia Xu, Alessandro Sordoni, Chandan Singh, Zelalem Gero, Michel Galley, Xingdi Yuan, and Jianfeng Gao. Test-time learning with an evolving library. *arXiv preprint arXiv:2605.14477*, 2026a.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026b.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *arXiv preprint arXiv:2504.01382*, 2025.

Ke Yang, Zixi Chen, Xuan He, Jize Jiang, Michel Galley, Chenglong Wang, Jianfeng Gao, Jiawei Han, and ChengXiang Zhai. Plugmem: A task-agnostic plugin memory module for llm agents. *arXiv preprint arXiv:2603.03296*, 2026.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. Assistantbench: Can web agents solve realistic and time-consuming tasks? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8938–8968, 2024.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21457–21483, 2026.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, volume 2024, pp. 19036–19066, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Table A.1: **Comparison with representative web and data analysis benchmarks.** We compare evaluation scale, access to live websites, source scope, analysis over record sets, and question density per source. Rows are grouped by benchmark type: Data, Web, Data + Web.

Benchmark	# Evaluation tasks	Live web	Scoped-source	Needs data analysis	Analytical QAs per source
Spider 2.0 (Lei et al., 2025)	632	✗	✓	✓	–
DSBench (Jing et al., 2025)	540	✗	✓	✓	–
DABstep (Egg et al., 2025)	450	✗	✓	✓	–
WebArena (Zhou et al., 2024)	812	✗	✓	✓	63.8
WebChoreArena (Miyai et al., 2025)	532	✗	✓	✓	104
Online-Mind2Web (Xue et al., 2025)	300	✓	✓	✗	–
WebLists (Bohra et al., 2025)	200	✓	✓	✗	4
WideSearch (Wong et al., 2026)	200	✓	✗	✗	–
DeepWideSearch (Lan et al., 2025)	220	✓	✗	✗	–
AssistantBench (Yoran et al., 2024)	214	✓	✗	✓	–
WebAggregatorQA (Wang et al., 2026)	159	✓	✗	✓	–
WebDS (Hsu et al., 2026)	870	✓	✓	✓	30
Ours	**3,799**	✓	✓	✓	**135.7**

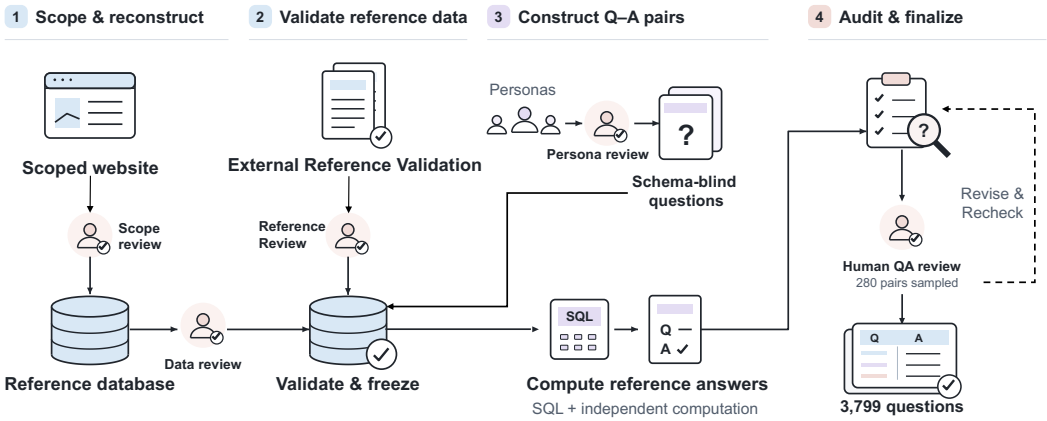


Figure 4: Workflow of Building DATAMEMOBENCH.

A DATAMEMOBENCH DETAILS

A.1 COMPARISON WITH EXISTING BENCHMARKS

Table A.1 compares DATAMEMOBENCH with representative web and data analysis benchmarks. Existing benchmarks cover complementary settings, including analysis over fixed datasets, interaction with controlled web environments, and information gathering from live websites. DATAMEMOBENCH focuses on deep, repeated analysis over real websites. Its tasks require agents to gather, compare, and aggregate large amounts of data records while navigating the complexity of live web sources. A dense set of analytical questions per source tests whether agents can build and reuse a detailed understanding of the underlying data across successive requests.

A.2 DESIGN PRINCIPLES FOR QUESTIONS AND REFERENCE ANSWERS

DATAMEMOBENCH's construction workflow is illustrated in Figure 4. We follow seven principles when writing questions and computing reference answers.

Realism. Each question should be something one of the personas would plausibly ask, such as an esports analyst comparing champion classes. Persona generation and question generation run as separate stages, and both see only an overview of the website, never the reference database or its schema, so questions follow what simulated users want to know rather than what the columns contain.

Objectivity. Each answer must follow from the data, not from opinion. Terms like "a major studio" are allowed only with an explicit definition, such as "studios with at least five nominations". We also exclude "why" questions and categories taken from outside editorial taxonomies, since the source has no field for either. Reference answers are numbers, short lists, or short factual statements, so grading compares values instead of judging prose.

Clarity. Each question should admit only one reading. Where they affect the answer, it states the denominator of any rate, how entities are matched and counted, rounding, how missing values are handled, and how ties are broken. Ambiguous questions are caught by the blind answering pass and the premise audit (Appendix A.6), and we manually rewrite or remove every question they flag.

Verifiability. Reference answers are computed by scripts, not written by a model, and record how they were derived, for example "San Mateo: 0.0045 (29/6390)".

Context independence. The agent sees one batch of questions at a time, so each question must make sense on its own. Questions do not point to other questions ("in that same ranking"), leave a set unlisted ("both regions"), or use phrases like "this paper".

Time invariance. Answers must not change over time. Questions name a year or version instead of saying "current" or "latest", and we do not ask about values that keep changing, such as prices or live rankings.

Diversity. We avoid filling the set with one template and different parameters. After numbers are replaced with a placeholder, two questions whose wording overlaps by 80% or more count as near duplicates. Each cluster of near duplicates holds at most two questions, and at most 10% of answerable questions may belong to any cluster. The same check confirms that the answer stage never edits question text.

A.3 Question Generation

Persona and question generation are guided by two separate Claude Code skills running Claude Opus 4.8. Both receive the same overview of the source, written in domain vocabulary; neither can inspect the reference database, its schema, the answer scripts, or any reference answer.

Personas. The persona prompt (Box A.1) receives only the overview and returns 5 to 10 personas. Each testbed receives a median of 8 personas.

Box A.1 · Persona generation prompt, identical for every testbed

```
OVERVIEW
{{OVERVIEW}}

You are given a description of what the website contains (the OVERVIEW). Your job is to
describe the people who read this site: write 5 to 10 personas, each a plausible user
with a real reason to visit it.

The OVERVIEW is your only input. Do not open any file, directory, database, schema,
script, or result folder, and do not ask for a sample of the data.

Each persona has a name and a description. Write the description in the second person
("You are ..."), in two to four sentences that state:
(1) who you are: your background and role;
(2) the concrete goal that brings you to the site;
(3) what you care about most in the information you find there.

Make the personas differ from one another in role, level of expertise, and motivation,
```

```
so that together they cover the range of people who actually read the site.

Describe needs in the words a reader of the site would use. Do not mention databases,
tables, fields, columns, queries, storage, or anything else about how the site is built.

Every need must be one the OVERVIEW says the site can serve. Do not invent needs for
data the site does not publish.

Output
Return one JSON document and nothing else:
[{"name": "...", "description": "You are ..."},
 ...]
```

Questions. The question prompt (Box A.2) receives the overview and the personas and returns question text only, with each question tagged by its persona, its label (`standard` or `hard`), and its intended pattern. Standard questions cover lookups, filters, aggregations, and descriptions of single records, varied in shape. Hard questions must use at least one pattern beyond a plain chain of filtering, grouping, aggregation, and extremum, with at least one step consuming a value produced by an earlier step. The prompt lists nine such patterns and requires each of the three most demanding (multihop across entities, negation or absence, and top k per group) at least twice. Questions are then screened for near duplicates as described under the diversity principle. Unanswerable probes are constructed separately (Appendix A.4).

Box A.2 · Question generation prompt, identical for every testbed

```
OVERVIEW
{{OVERVIEW}}
PERSONA
{{PERSONAS_JSON}}

You are given a description of what the website contains (the OVERVIEW) and a persona
who uses the site. Your job is to write question text only: no answers, no queries. A
separate, later step will compute every answer against the catalog; you never see that
step and it never changes your wording.

You have exactly two inputs: the OVERVIEW, the PERSONAS. Do not open any file,
directory, database, schema, script, or result folder. Do not search the repository. Do
not ask for a sample of the data. If you find yourself wanting to know a field name, a
stored value, or the exact size of something, that is the signal to phrase the question
in the domain vocabulary a person browsing the site would use and let the later step
handle storage.

Standard questions:
Lookups, filters, aggregations, descriptions of one record across several attributes:
the everyday traffic the site serves. Each is recognisably its persona's question. Vary
the shape: a count, a which, a value for a named entity, a yes/no with a stated rule, a
short list with a stated ordering, a legitimate "none" with the empty branch offered.

Hard questions:
Hardness is composition, not length. Every hard question must use at least one pattern
beyond a plain chain (filter, then group, then aggregate, then extremum), and at least
one step must consume a value produced by an earlier step (an aggregate, a rank, a
derived grouping). Distribute across these patterns, and use each of the three starred
ones at least twice:

Pattern: shape (in domain words)
Nest: an aggregate over a set that is itself defined by an aggregate ("among sectors
  whose average ... exceeds ..., which ...")
Self join / graph: turn a listed attribute (authors, tags, instructors) into the unit of
  analysis, then group or traverse by it
Set compose: intersection / difference / union of two independently derived populations
Extremum of extrema: the group whose per group maximum/minimum is itself the
  largest/smallest
Group level correlate: aggregate per group, then compare or rank the group level
  aggregates
Counterfactual: recompute a figure with a stated removal or reclassification applied
* Cross entity multihop: traverse three or more relational levels (record, then a listed
  person, then that person's other records, then a property of those)
* Negation / absence: which groups or entities have no matching record; always name the
  full universe ("of the 21 subject areas ...") so the answer is a concrete list
* Per group top k: for each of the top N groups, the top k members; an answer with
  multiple rows, with N, k, ranking key and tie break all stated
```

```
Output
Return one JSON document and nothing else:
{"testbed": "{{TESTBED}}",
 "questions": [
   {"id": "{{ID_PREFIX_STANDARD}}{{ID_START_STANDARD}}", "question": "...",
    "persona": "<persona name>", "label": "standard",
    "intended_pattern": "lookup|filter|aggregate|extremum|list|yes_no|empty"},
   {"id": "{{ID_PREFIX_HARD}}{{ID_START_HARD}}", "question": "...",
    "persona": "<persona name>", "label": "hard",
    "intended_pattern": "nest|self_join_graph|set_compose|extremum_of_extrema|
     ↪ group_level_correlate|counterfactual|cross_entity_multihop|negation_absence|
     ↪ per_group_topk"}
 ]}
```

A.4 Unanswerable Probe Construction

Unanswerable probes test the agent's hallucinations. A probe is valid only if the source itself lacks the requested information, in which case a refusal is graded as correct.

Kinds of absence. Probes are spread over four kinds of absence (Table A.2), so that an agent cannot learn a shortcut such as treating every question about a particular attribute as a trap.

Table A.2: The four kinds of unanswerable probes, with examples from the IRS and NVIDIA testbeds. In each case the source offers a tempting substitute, but the correct answer is that the source does not publish the requested value.

Kind	Example probe	What the source publishes	Why an agent answers anyway
Absent attribute	How many dependent children were claimed on Ohio returns in tax year 2023?	Counts of *returns* claiming child credits, but no count of children.	A nearby count looks usable, but one return can claim several children.
Absent entity	How much onboard memory does the NVIDIA RTX A4800 professional graphics card carry?	The RTX Ampere professional line: A400, A1000, A2000, A4000, A4500, A5000, A5500, A6000. The catalog lists no A4800.	The name fits the series pattern, so a value can be interpolated from its neighbors.
Absent relation	What share of returns that moved from New York to Florida in 2022–23 were filed jointly?	Migration flows and filing status, but no table that combines them.	Both halves are in the data, so the filing status mix of all residents can be applied to movers.
Absent granularity	How many returns were filed in California in each calendar month of 2023?	One annual figure per place and tax year.	The annual total can be split evenly across twelve months.

Admission checks. Each candidate must pass four checks before it enters the question set.

(1) Absent from the source. We search the source exhaustively and make sure the question is not answerable from the source.

(2) Not derivable from published fields. The value must not be computable from fields the source does publish.

(3) Indistinguishable from ordinary questions. A probe matches ordinary questions in tone and length and is attributed to a persona, so an agent cannot spot it from its wording.

(4) A correct rationale. The reference answer states why the value is unavailable, and we check that reason as well as the verdict, because the judge reads it when grading.

Valid zero or empty set answers are not probes; such questions remain answerable.

Table A.3: Examples of published facts used to validate the reference databases.

Website (publisher)	Question	Published answer
IRS SOI (Tax Foundation)	What was New York's net migration of tax returns in the 2022–23 IRS migration file?	−71,987
America's Health Rankings (KFF)	Among states ranked for smoking in data year 2024, which has the highest adult smoking prevalence?	West Virginia (20.78%)
NVIDIA (Tom's Hardware)	Of the GeForce GTX 1660, 1660 SUPER and 1660 Ti, which has the highest rated board power?	GTX 1660 SUPER (125 W)
arXiv (Crossref)	How many authors are listed on "Dimensional Analysis in Economics: . . . "?	3
Steam (Metacritic)	What is the Metacritic Metascore of the PC version of *Objects in Space*, as recorded on 2026-09-01?	79

A.5 Reference Database Construction and External Reference Validation

Construction. Reference databases are reconstructed from cached source responses using versioned scripts. Checks include reconciliation against totals the source reports where available, duplicate detection, coverage of each field, and consistency constraints specific to each domain. Missing values are distinguished from extraction failures and source placeholders. The authors review each schema, record counts, missing values, and the outcome of these checks.

Sourcing published facts. A separate Claude Opus 4.8 sourcing agent sees only the source name, URL, and domain, not our database. It collects facts that other publishers have reported about the source (Table A.3) and records each with its publisher, URL, retrieval time, and the text it read. We aim for at least three publishers and 100 facts per testbed, with no publisher contributing more than about 70%. Some publishers rework the same primary data as the website, such as the Tax Foundation and KFF rows; agreeing with them checks our extraction and arithmetic, not the original data. Others are independent, such as the Crossref row, whose author list the journal publisher deposited separately from arXiv.

Development and test splits. The 3,508 collected external QAs are split roughly in half into development (1,763) and test (1,745) sets, stratified by question family with a fixed seed. A separate agent answers them from the reference database alone, without network access or the published answers, and reports when data are missing. We use the development split to find and fix extraction errors: each mismatch is traced to an incorrect stored value, missing coverage, or a question outside the source's scope, and the authors resolve ambiguous questions or wrong published answers. We then freeze the database and run the test split once. The frozen databases answer 1,712 of the 1,745 test questions correctly (98.1%) across the 21 testbeds with external references. None of the 33 mismatches is caused by a wrong database value. After validation, we found that most are failures of the answering agent or its environment, such as rate limits or giving up after a single query. The rest come from sources that disagree with each other (for example, Steam and Metacritic show different Metascores for the same game), from the grader's matching rules, or from question wording and premises. We fix the grader and reword or remove the affected questions, without changing any database or reference answer.

Additional checks. Where available, we also compare database values against independent calculations from source APIs. For the 7 testbeds without independent reporting, the authors manually check the schema, the completeness of the collection, and the accuracy of stored values.

A.6 Reference Answer Computation and Checks

Computation. Reference answers are computed by scripts under the verifiability principle (Appendix A.2) and checked through a second, separately written query or an alternative computation. Each reference retains its derivation, including tied answers and the records supporting any absence claim. Unanswerable probes additionally pass the admission checks in Appendix A.4.

Table B.1: Memory conditions under a shared interface. Procedural and episodic memory use the same textual format; continued session is a reference condition.

Condition	$\mathcal{M}$	**READ**	**WRITE**
No memory	$\varnothing$	—	—
Textual memory			
Procedural	$\mathcal{N}$	inject notes	update notes
Episodic	$\mathcal{E}$	retrieve episodes	update episodes
Data (ours)	$(\mathcal{D}, \mathcal{N})$	notes, schema; SQL	evolve on miss/stale
Continued (ref.)	context	prior dialogue	append turns

Blind checks. Claude Opus 4.8 agents perform two complementary blind checks. An answering pass attempts each question from the reference database without seeing its reference answer. A premise audit checks population boundaries, matching conventions, ties, degenerate superlatives, and whether incomplete acquisition could change the answer; each reported defect must be supported by a database query. Agreement between computations is treated as a consistency check rather than sufficient evidence that a question is valid: grading compares an answer with its reference, so two computations of the same degenerate quantity agree perfectly.

Resolution. Identified issues are resolved manually by correcting answers, revising questions, or removing unsupported questions, followed by revalidation.

A.7 VERSIONING AND LIVE SOURCE CHANGES

Why reference answers remain valid on live sources. Evaluated agents read the live websites, whereas reference answers are computed from cached snapshots. Three design choices keep the two consistent: selection criterion (4) excludes collections whose answers change constantly; the scope of each testbed is fixed before any question is written; and the time invariance principle restricts every question to a closed scope, such as a completed event, a fixed historical period, or a named version, and rules out values that drift (Appendix A.2). Records inside such a scope are no longer updated in the normal course of the site's operation, so an agent reading the live site at evaluation time sees the same records as our snapshot. For example, ICLR questions concern the completed 2026 review cycle and name the conference year.

Versioning. Cached source responses, build scripts, reference derivations, and versioned question files support reproducibility and later inspection. Explicit dates, versions, and population boundaries reduce temporal ambiguity but do not freeze the live websites used during evaluation. We therefore distinguish source changes from extraction errors and reference defects. Revisions retain prior artifacts and record the affected questions and the reason for each correction or retirement. Corpus statistics and reported scores refer to their corresponding question versions.

B DATAMEMO IMPLEMENTATION DETAILS

B.1 MEMORY CONDITIONS

Because $\mathcal{M}$ is the only channel between batches, the forms we compare are settings of the interface above, differing only in what $\mathcal{M}$ holds (Table B.1); model, questions, order, judge, and storage cap are identical. Procedural memory retains reusable strategies and rules (Ouyang et al., 2026; Zhang et al., 2026), whereas episodic memory retains task-specific experiences (Zheng et al., 2024). Both are implemented within our shared harness. DATAMEMO combines queryable relational records with procedural notes. We also report a *continued-session reference* that retains full context (including conversation and working directory) across batches within the context limit.

Table B.2: Agent configuration. Every value is identical across the five conditions of a base model.

Framework	`smolagents` 1.26 `ToolCallingAgent`, default system prompt
Base models	GPT-5.5 (Azure OpenAI, `gpt-5.5-2026-04-24`); Claude Opus 4.8. Reasoning effort medium
Tools	`shell`; `fetch` (URL or ordered URL list; cloud browser); `visit_webpage`, `web_search`
Question stream	batches of 5 questions, one fixed order per testbed (seed 20260830), at most one unanswerable probe per batch
Time budget	2,700 s per batch; 10,800 s per condition.
Storage cap	2 GiB on the memory directory, enforced by the memory scripts.
Isolation	the shell runs inside a Landlock filesystem fence: it can write only its own run directory
Answer visibility	none; the agent never sees a grade or a gold answer

B.2 Shared across all conditions

Every condition runs the same `smolagents` `ToolCallingAgent` with the same four tools, the same budgets and the same question order. The memory tools are not extra agent tools: they are Python scripts placed in the working directory and invoked through `shell`. The action space is therefore identical in every condition; only the scripts present, and the prompt that documents them, differ. Each batch is a single answering phase. There is no separate reflection pass, so a condition pays for memory maintenance only with the turns it spends while answering, inside the same budgets as the others.

Every batch prompt of every condition is assembled from the same parts in the same order (Box B.1). Only the block marked **<MEMORY BLOCK>** depends on the condition. The source map is deliberately minimal: it names the site and one entry URL and states that the open web may be used. What the site does and does not publish, and how to reach it, is exactly the knowledge a memory is supposed to accumulate, so none of it is given to any condition. The environment line names only the scripts actually present in the condition's working directory.

Box B.1 · Batch prompt, identical in every condition

```
Answer these {site_label} benchmark questions (batch {k}). There are several batches in
total.
Answer them efficiently.
Reuse what you already have in memory instead of redoing.

INTEGRITY RULES (critical):
- Do NOT access the project repository in any way. [...] Stay in your current working
  directory.
- Never look for or read gold answers, question files, or any prior experiment outputs.

You may use the live site and the open web to gather the data.
Live site -- {site_label}: {entry_url}

ENVIRONMENT: use `python3` [...]; names only the scripts staged for this condition.
RUN EVERYTHING SYNCHRONOUSLY IN THE FOREGROUND: never launch a command in the background
[...]

<DATA-HANDLING RULES, Box B.2>
<MEMORY BLOCK>   <- the only condition-specific part (Section B.3)

Return exactly one JSON object and no prose outside it, of this shape:
{"records": [{"id": "question id",
              "answer": "final answer, or null",
              "status": "answered | unanswerable | error",
              "method": "brief description of how you got the answer"}]}
Preserve each `id` exactly.

Questions:
1. [{qid}] {question}
...
5. [{qid}] {question}
```

Box B.2 · Data handling rules, every condition

```
THE SOURCE
* ANSWER FROM THE NAMED SOURCE, NOT THE WEB. The source is what the source map above
  names -- its site, its API and any bulk surfaces it offers. The open web may help you
  REACH and UNDERSTAND that source (how an endpoint is called, what a field encodes); it
  is not a second place to take values from. Every value you answer with must trace to
  records fetched from the named source.
  When a question asks for something that source does not carry -- a field it has no
  place for, an entity it does not list -- that absence IS the answer: answer with
  status
  "unanswerable". Do not fill the gap with a figure from another site.
* KEEP THE SOURCE'S OWN WORDS. A question that asks which label, category or phrase
  something carries is asking for the source's wording, exactly as displayed, including
  any qualifier or prefix.

COMPLETENESS
* PAGINATION / LAZY LOAD: a listing that returns exactly a round size (25/50/100), or a
  grid/feed that loads more on scroll or takes a `?page=`/offset param, is almost
  certainly showing only the first chunk. Keep paging and re-counting until the total
  STOPS GROWING.
* CHECK ONE COUNT YOU CAN SEE. If the source states a total anywhere, compare your
  record count to it; if you are short, fetch the missing records. A check that recounts
  the records you just parsed verifies nothing: it takes two independent derivations of
  the same quantity.
* SPEED: fetch many pages CONCURRENTLY (a small thread pool, ~4-8 workers); a serial
  per-page loop is the main time sink. Run it synchronously in the foreground.

COMPUTING
* ANSWER SHAPE: compute each answer with code over the records (count, sum, average,
  sort, take the top), not by reading a printed list. If a question asks for an entity
  AND a figure, give both.
```

The `status` field lets an agent report a recognised absence (`unanswerable`) rather than guess; together with the first rule of Box B.2, every condition is told the same thing about when absence is the answer, and it is graded the same way in every condition.

B.3 Memory in each condition

Four of the five conditions start every batch in a new session whose working directory has been wiped down to `./mem`; for them, memory crosses a batch boundary only through files in `./mem`, and the harness deletes any file there that the condition is not allowed to keep. Continued session is the exception: it continues one conversation.

B.3.1 No memory

The **<MEMORY BLOCK>** is empty on every batch. Nothing is staged and nothing survives the wipe, so every batch starts from the shared prompt alone.

B.3.2 Procedural

Box B.3 · Procedural memory, batch 1

```
MEMORY TOOL -- textual notes (note.py). Your memory persists ONLY in ./mem; each batch
is a brand-new session with NO memory of this conversation, and the working dir is wiped
to ./mem between batches, so anything outside ./mem is LOST.

  python3 note.py add --text "..." [--tags a,b]   # save a reusable note
  python3 note.py list | read --id N | update --id N --text "..." | delete --id N

Your notes are the ONLY thing that survives between batches. Use them to make each batch
faster and more accurate than the last. Store two kinds of knowledge: (1) data ACCESS --
how to reach/fetch the data (endpoints, parameters, pagination patterns, rate-limit
gotchas, dead ends -- include WORKING CODE when a procedure took effort to discover);
(2) data PROCESS -- how to compute answers (matching/dedup/counting conventions, field
meanings, pitfalls). A note write that rides along in a turn you are already taking
costs nothing -- never skip one to save a tool call.

HOW TO WRITE NOTES (a rulebook, not a journal):
  1. ATOMIC: one note = one reusable convention, stated once.
```

```
  2. RULE, NOT ANSWER: store only the transferable METHOD that RE-DERIVES an answer. A
     note that names a specific record/entity or pins a specific count is WRONG; rewrite
     it as the general rule.
  3. CONSOLIDATE, BUT DO REVISE: fold a new convention into the note that covers that
     topic [...]
  4. WHEN TO WRITE ONE: WHERE THE DATA LIVES (incl. dead ends) and WHICH READING IS
     RIGHT [...]
  5. ALWAYS, AFTER A FETCH: record or update the access note before you finish.
  6. TEMPORAL PROVENANCE: record when source-dependent access or interpretation
     rules were last checked (UTC), and any source-provided as-of date or
     reporting period relevant to them. These timestamps are allowed metadata,
     not stored answers. Preserve them when reusing a note.

Shape: "[TOPIC] rule: <one-line rule>. gotcha: <one pitfall, if any>."
ACCESS IS THE EXCEPTION: when reaching the data took a PROCEDURE, the note body is the
WORKING CODE, verbatim [...]
```

Box B.4 · Procedural memory, batch ≥2

```
MEMORY (note.py) -- your notes from earlier batches are inlined below. Each batch is a
fresh session; only ./mem survives.

BEFORE YOU WORK: read through your notes. They contain the access recipes and
interpretation rules you worked out previously.
Everything below is your own earlier work and has not been checked against any answer
key. Reuse it when it covers the current question. Go back to the source when a question
depends on something it does not hold.

  python3 note.py add/list/read/update/delete
      # refine a recipe note only if you learned something new
<HOW TO WRITE NOTES: same six rules as Box B.3>

This is a fresh session, but your memory from earlier batches is on disk in ./mem and
its CURRENT contents are inlined IN FULL below. Use this as your starting point.

NOTES (id: full text; these are ALL your notes, so do NOT run `note.py list`/`read`):
  [{id}] {note text}
  ...
  ...
```

B.3.3 EPISODIC

Box B.5 · Episodic memory, batch 1

```
MEMORY TOOL -- episodes (episode.py). Your memory persists ONLY in ./mem; each batch is
a brand-new session with NO memory of this conversation, and the working dir is wiped to
./mem between batches, so anything outside ./mem is LOST.

  python3 episode.py log --question "..." --answer "..." [--qid X] [--steps "..."]
        [--sources url1,url2] [--outcome "..."] [--tags a,b]   # record one case
  python3 episode.py search --query "..." [--n 5]
      # the past cases most similar to a question
  python3 episode.py list | read --id 3,7 | delete --id N
  python3 episode.py update --id N [--answer ...] [--outcome ...]

Your episodes are the ONLY thing that survives between batches. Use them to make each
batch faster and more accurate than the last. Each episode is a case file recording one
question you handled: what you were asked, what you did, what you answered, and whether
it worked.
An episode write that rides along in a turn you are already taking costs nothing --
never skip one to save a tool call.

HOW TO WRITE AN EPISODE (a case file, not a rulebook -- one per question you handle):
  1. ONE QUESTION = ONE EPISODE, logged as soon as you have answered it.
  2. KEEP THE SPECIFICS: the question, the answer you gave, the exact route (endpoint,
     parameter, filter), and the key data values you found. Specific values are what
     make an episode directly reusable without re-fetching.
  3. SAY HOW IT WENT: what was slow, what returned nothing, where you guessed.
  4. REVISE, DON'T REWRITE HISTORY: if an earlier answer was wrong, `update` its
     outcome.
  5. TEMPORAL PROVENANCE: in --steps, associate each source and its key values
     with their retrieval time (UTC) and any source-provided as-of date or
     reporting period. When reusing an earlier episode, retain the original
```

```
    timestamps; the new episode's creation time is not a new retrieval time.
```

Box B.6 · Episodic memory, batch ≥2

```
MEMORY (episode.py): the cases you handled in earlier batches. An index is inlined
below; the store may hold more than the index shows, so SEARCH first.
<COMMANDS AND HOW TO WRITE AN EPISODE: same as Box B.5>

BEFORE YOU WORK: search your episodes for each question, and log every question you
answer as a new episode.
Everything below is your own earlier work and has not been checked against any answer
key. Reuse it when it covers the current question. Go back to the source when a question
depends on something it does not hold.

This is a fresh session, but the record of your earlier batches is on disk in ./mem.
Start from it.

EPISODES: {n} past case(s); the {m} most recent are indexed here (id, batch, question,
answer):
  [{id}] (b{k}) {question, truncated} → {answer you gave}
  ...
```

B.3.4 DATAMEMO

Box B.7 · Data memory, batch 1

```
MEMORY TOOLS -- two stores, both persisting ONLY in ./mem. Each batch is a fresh session
[...]
A memory write that rides along in a turn you are already taking costs nothing -- never
skip one to save a tool call.

(a) note.py -- textual GUIDELINES memory: data ACCESS (how to fetch) and data PROCESS
    (matching/counting conventions, gotchas). Your table schemas are shown to you
    automatically on every later batch -- do not spend notes indexing them.
    <HOW TO WRITE NOTES: same six rules as Box B.3>

(b) table.py -- structured DATA memory. Your store is mem/memory.sqlite: the tables you
    build plus a ledger of where each came from.

  python3 table.py run --code -   # Python against your memory. Pre-opened:
      #   conn       -- sqlite3 to memory.sqlite
      #   coverage(table, note, total=N, total_from=...) -- how COMPLETE it is
  python3 table.py query --sql "SELECT ..."   # answer a question -- capped at 20 rows
  python3 table.py exec | inspect

TABLE DESIGN: store the source as it is, so later batches can answer new questions.
  * ONE ENTITY = ONE TABLE of RAW rows, not aggregates; each multivalued field goes in
    one child table.
  * the source's own words go in a VERBATIM column; a parsed value goes in a SEPARATE
    column beside it.
  * absence is NULL. A field the source does not carry gets a dead end note, not a
    filled column.
  * variants and unranked entries are stored so that one listed entity is one base row.

SAVE THE DATA YOU FETCH: put the records into your tables through table.py, then answer
from them with a query that returns the answer.
```

Box B.8 · Data memory, batch ≥2

```
MEMORY (note.py) -- your notes from earlier batches are inlined below. They are the half
of your memory the tables cannot hold: WHERE a source lives, which endpoint carries
which field, the reading you settled on. [...]

MEMORY (table.py) -- the tables below already exist in mem/memory.sqlite, each shown
with its build age and any source you recorded. QUERY:

  python3 table.py query --sql "SELECT ..."
  python3 table.py run  --code -

BEFORE YOU WORK:
```

```
Everything below is your own earlier work and has not been checked against any answer
key. Reuse it when it covers the current question. Go back to the source when a question
depends on something it does not hold.

This is a fresh session, but your memory from earlier batches is on disk in ./mem and
its CURRENT contents are inlined IN FULL below.

TABLES (name: columns [row count] (build age; source; coverage)):
  {table}: {col}, {col}, ...  [{n} rows]  ({age}; source: S1; coverage: {text})
  ...

NOTES (id: full text -- these are ALL your notes; do NOT run `note.py list`/`read`):
  [1] {note text}
  ...
```

B.3.5 CONTINUED SESSION

Box B.9 · Continued session, batch 1

```
MEMORY: this conversation continues across all batches. Each later batch arrives as a
new message in this same session, and this working directory is never wiped, so any
file you write here stays available to later batches. If the conversation grows too
long, it will be compacted: you will then see a recap of your earlier answers and the
methods you reported instead of the conversation, and your files will remain.

Use what persists to make each batch faster and more accurate than the last. A file
write that rides along in a turn you are already taking costs nothing; never skip one
to save a tool call.
```

Box B.10 · Continued session, batch ≥2

```
# Resumed session:
# the shared blocks and Box B.9 are already in the conversation.
Same session as the previous batches. Your earlier work in this conversation and in
this working directory has not been checked against any answer key. Reuse it when it
covers the current question. Go back to the source when a question depends on
something it does not hold. Answer ONLY the questions below.

# After a batch whose input exceeded the token limit:  a NEW session; the full shared
    prompt
# (source map, Box B.2) precedes this block.
This is a FRESH session: your earlier conversation was compacted. Your memory is the
SUMMARY below plus whatever files you left in this working directory; do not assume any
live conversation state. Later batches continue this new session, and it will be
compacted again if it grows too long; your files remain. Answer ONLY the questions below.

Everything below is your own earlier work and has not been checked against any answer
key. Reuse it when it covers the current question. Go back to the source when a question
depends on something it does not hold.

SUMMARY OF YOUR EARLIER ANSWERS (every prior question, with the answer and the method
you reported; the work behind them is not included):
  [{qid}] {answer you gave}  [{method you reported}]
  ...   # every prior question, all batches
```

C ADDITIONAL RESULTS AND ANALYSIS

We provide further analysis of the results in this section.

C.1 EFFICIENCY AND DATA REUSE

Figure 5 compares accuracy against cost per correct answer. Within each base model, DATAMEMO achieves higher accuracy at a lower cost per correct answer than every baseline. Its efficiency advantage therefore persists when spending is measured relative to successful answers, rather than total cost alone.

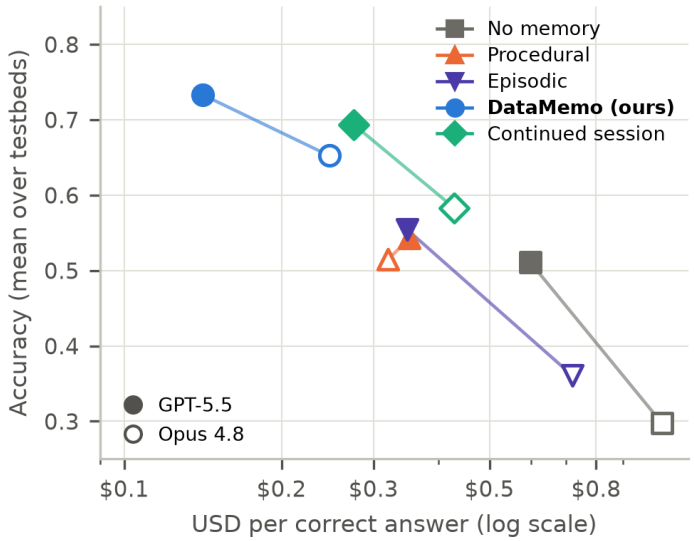


Figure 5: **Accuracy and cost per correct answer.** Filled and hollow markers denote GPT-5.5 and Opus 4.8, respectively. The cost axis is logarithmic. Lines connect the same memory condition across models.

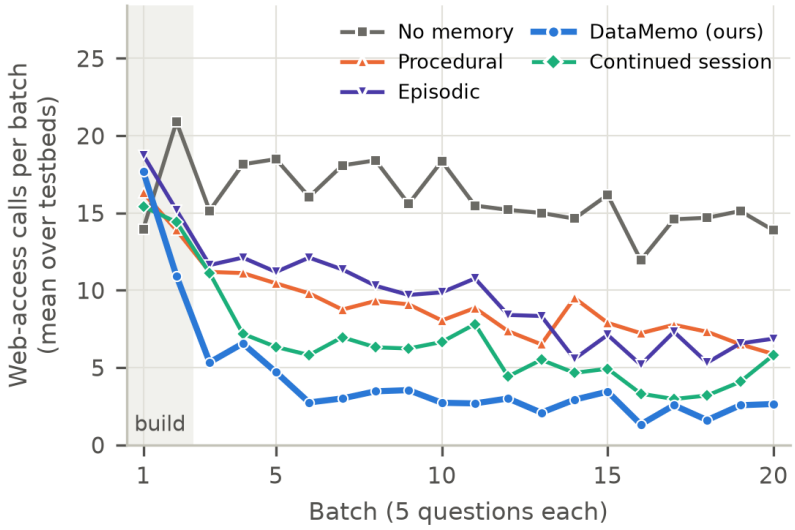


Figure 6: **Web access calls over successive batches.** Mean web-access calls per batch across testbeds using GPT-5.5. Each batch contains five questions.

Reduced web access in later batches. Figure 6 shows that web-access calls fall sharply for DATAMEMO over the initial batches and remain low thereafter. Continued session also reduces web access, but generally requires more calls than DATAMEMO, while no memory continues to access the web frequently throughout the stream. This pattern is consistent with reuse of accumulated records reducing repeated acquisition. The remaining calls can support acquisition of missing data and freshness checks; call counts alone do not distinguish these operations from redundant retrieval.

C.2 CASE STUDIES: HOW RETAINED INFORMATION SHAPES LATER ANALYSIS

We examine four paired execution traces to illustrate how different memory representations affect later analytical requests. Each pair receives the same five questions but carries the state produced by its own earlier batches. Table C.1 highlights one question from each pair. The cases illustrate

repeated acquisition, changes in data interpretation, and incomplete reuse of retained files; they do not estimate the prevalence of these behaviors.

No memory: rediscovery can change the data's granularity. In the IRS case, both agents compute a ratio of charitable-deduction returns to all returns, but over different income partitions. The no-memory agent uses state-summary rows from a county file whose highest band combines all incomes of at least $200,000. DATAMEMO queries a retained state table that separates this range into three bands, including incomes of at least $1 million. The different band and place returned by the baseline follow from the choice of source table, rather than the ratio formula. This case illustrates why retaining the source's granularity matters alongside retaining its values.

Procedural memory: knowing the route does not retain the records. The Stanford baseline already has notes describing the XML endpoint, department filters, and counting conventions. These notes guide a successful reconstruction of the focal answer, but the agent still crawls all 253 departments in the current batch. DATAMEMO obtains the same grading-basis result by querying its existing catalog. This pair illustrates an acquisition-cost difference even when the focal answers agree: procedural guidance preserves how to collect data, while retained records support computation without repeating that collection.

Episodic memory: repeated parsing can change a field's meaning. In the arXiv case, the episodic agent retrieves earlier answers and methods, then re-fetches bibliographic and raw metadata. Its final computation groups submitters by the bibliographic `created` date, whereas DATAMEMO uses stored version-1 dates. The traces expose this distinction for the same e-print, `1603.03675`: its bibliographic creation date is 2016-08-22, but its version-1 date is 2016-03-11. Thus, the fields are not interchangeable for first-posting queries. The retained version relation and accompanying interpretation note allow DATAMEMO to apply the intended temporal definition.

Continued session: retained files can cover the wrong population. The ICLR baseline retains both submission and review files after conversation compaction. However, it searches a rejected-only review file when the question asks about *all non-accepted* submissions, thereby omitting desk-rejected papers from the maximum-rating calculation. DATAMEMO expresses the requested population by joining submissions and reviews and excluding accepted submissions. The same coverage gap appears in another question in this batch: both agents identify 620 accepted papers whose ratings are all at least 6, but continued session uses 632 qualifying papers as the denominator, compared with 657 for DATAMEMO. This case illustrates that retaining files alone does not ensure that their combined coverage matches a later question.

Together, these cases illustrate the value of preserving records at the required granularity, keeping their interpretation explicit, and querying populations that match the current request. They also show that the benefit is not limited to correcting answers: the Stanford pair reaches the same focal result through substantially different acquisition paths. The observed timings include model and tool overhead and reflect each condition's accumulated state, so they should not be interpreted as isolated causal effects of representation.

C.3 ROBUSTNESS TO CHANGES IN WEB SOURCES

Updating current information and retaining historical evidence. Part A requires updated information available from the current website, whereas Part B asks about changes over a time window. DATAMEMO remains competitive on Part A, but its larger advantage appears on Part B, where retaining earlier records supports comparisons across time.

For sources that remove historical records, some Part B questions require evidence retained from the earlier observation; access to the current website alone need not suffice. When sources publish dated records, agents may instead recover that evidence by revisiting the website.

C.4 SENSITIVITY TO QUESTION ORDER

Table C.3 reports full results for the default order and two random shuffles with GPT-5.5.

Table C.1: **Paired cases of data reuse across successive requests.** Questions are abbreviated; answers and counts are taken from the traces. Batch time covers all five questions, including work within the batch, but excludes memory construction in earlier batches.

Task	Baseline execution	DATAMEMO execution
IRS, batch 17 Find the income band with the highest charitable-deduction share, then the place with the highest share within that band.	**No memory.** Downloads county data and uses its state-summary rows, with eight income bands. *Output:* $\geq$ \$200,000; Utah; 51.46%. *Batch time:* 1,741.1 s.	Queries the retained state table, which distinguishes ten income bands. *Output:* $\geq$ \$1,000,000; District of Columbia; 79.23%. *Batch time:* 27.0 s.
Stanford, batch 29 Find the most common grading basis in each of the four subjects with the most undergraduate courses.	**Procedural memory.** Follows stored XML-access notes and crawls 253 departments, retrieving 15,842 course nodes. *Output:* Same four subjects, grading basis, and counts as DATAMEMO. *Batch time:* 256.8 s.	Queries 15,906 retained course rows and groups undergraduate courses by subject and grading basis. *Output:* Letter or Credit/No Credit for all four; HISTORY 281, MUSIC 219, ENGLISH 219, CSRE 132. *Batch time:* 93.3 s.
arXiv, batch 11 Identify the top two submitters by first-posting year (2025 shown from a three-year query).	**Episodic memory.** Retrieves prior episodes, re-fetches metadata, and groups records using the bibliographic `created` year. *Output:* Masahiro Kato (17); Tatsuru Kikuchi (16). *Batch time:* 210.7 s.	Joins retained records to version dates and groups by the version-1 year, following its stored interpretation rule. *Output:* Georgy Lukyanov (25); Tatsuru Kikuchi (17). *Batch time:* 97.8 s.
ICLR, batch 8 Find the highest mean reviewer rating among all non-accepted submissions and report the venue status.	**Continued session.** Reuses local files after compaction, but searches only the rejected-submission review file. *Output:* 7.0; rejected submission. *Batch time:* 70.8 s.	Joins submissions and reviews, excludes accepted submissions, and retains other non-accepted statuses. *Output:* 8.0; desk-rejected submission. *Batch time:* 63.2 s.

Table C.2: **Change-question accuracy per site.** Part-B accuracy with the sites split by whether the source publishes its own dated history. Bold marks the best in each column.

	publishes no history			*publishes history*	
Memory	Y Comb. *75*	NYSE *37*	Apple *22*	League *96*	arXiv *51*
No memory	.29	.27	.55	.46	.71
Procedural	.29	.35	.50	.35	.75
Episodic	.29	.41	.73	.47	**.76**
Continued session	**.87**	.62	.95	.43	.63
DATAMEMO	.85	**.78**	**1.00**	**.58**	.75

Table C.3: **Sensitivity to question order.** Results for the default order and two random shuffles. Only the strongest baseline, continued session, is rerun due to compute limits. Shaded rows report mean ± SD and coefficient of variation (CV) over the three orders. Cost and time are per testbed.

	DATAMEMO			Continued session		
Order	Acc. ↑	Cost ($) ↓	Time (min) ↓	Acc. ↑	Cost ($) ↓	Time (min) ↓
Original	0.733	14.1	93	0.693	25.9	107
Shuffle 1	0.735	15.4	89	0.731	27.0	103
Shuffle 2	0.726	14.8	90	0.695	26.1	101
Mean ± SD	0.731 ± 0.005	14.8 ± 0.7	90.7 ± 2.1	0.706 ± 0.021	26.3 ± 0.6	103.7 ± 3.1
CV (%)	**0.6**	4.5	2.3	3.0	2.3	2.9